# Java PathFinder
## Second Generation of a Java Model Checker

Guillaume Brat[1,3] and Klaus Havelund[1,3] and SeungJoon Park[2,3] and Willem Visser[2,3]

[1] RECOM Technologies
[2] Research Institute for Advanced Computer Science (RIACS)
[3] Automated Software Engineering Group
NASA Ames Research Center
{brat,havelund,spark,wvisser}@ptolemy.arc.nasa.gov

## 1 Motivation

Model checking is seldom applied to implementation programs. Furthermore, when it is applied, the usual approach is to extract relevant portions of the code, create a model of its behavior in a different notation, and then check the latter. This approach has the drawback that it requires expertise in the use of the model checking tools and hence will not, in general, allow software developers to check their own code during development. In the Automated Software Engineering group at NASA Ames we are investigating the use of model checking on actual source code in order to allow (NASA) software developers to augment their traditional testing techniques with model checking. Here we describe our work on applying model checking to Java programs.

The first Java model checker developed within the group, Java PathFinder 1 (JPF1) [10], used an automatic translation from Java to PROMELA (the input notation for the SPIN model checker) in order to do model checking. JPF1 was highly successful in finding errors in complex Java programs [11] since it covered a large part of the Java language, e.g. dynamic object and thread creation, exceptions, etc. However, it could not handle language features that were not supported by PROMELA, for example floating point numbers. Furthermore, since the tool was based on a translation from Java to PROMELA, it required all the Java source code for a program to be available, and this is often not the case when libraries are used. In order to solve these problems we built a new Java model checker, JPF(2), that works directly on Java bytecode. Note, although this work is related to what is commonly referred to as bytecode verification, which checks static security properties [2], we also check behavioral properties. In Section 2 we describe JPF's structure and some of its novel features.

Model checking often suffers from the state-explosion problem and when checking actual source code this problem is even more acute. We use abstraction (Section 3), static analysis (Section 4), and runtime analysis (Section 5), in order to alleviate some of the state-explosion problems.

## 2 Model Checking

JPF is an explicit-state model checker for programs written in Java. It is itself written in Java and contains a special-purpose Java Virtual Machine (MC-JVM) as its core. The major difference between MC-JVM and other JVM implementations is that a JVM suitable for model checking requires efficient memory management rather than blistering speed. Developing JVMs in Java is not new, two other examples exist of which we are aware, *Rivet* [3] and *Java-in-Java* (JIJ) [16], and interestingly their existence is also due to a desire from their developers to analyze Java programs in a more efficient way[1]. Neither implementation however considers model checking as the end goal: Rivet is used for advanced testing and JIJ for advanced debugging.

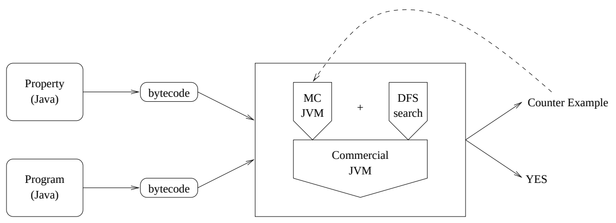

**Fig. 1.** JPF System Layout

Currently, JPF can only check invariants and deadlocks. An invariant is specified as a Java method that returns a boolean value — this method can examine the contents of the memory of the program being checked by using general Java statements and expressions as well as special-purpose methods provided by the MC-JVM. Figure 1 shows the system layout. Both the program to be checked and the property are translated to bytecode and are given as input to the model checker. The model checker consists of two parts: the MC-JVM that executes the bytecodes and the depth-first algorithm that does the actual traversal of the state-graph of the program. The depth-first algorithm can only tell the MC-JVM to move *forward* one step, move *backward* one step and *evaluate* the current invariant. The MC-JVM keeps track of whether states have been visited before, by storing all new states in a hashtable. It also stores the current path of states in a stack to allow backtracking (i.e. when it receives the command to move backward). Both the MC-JVM and the traversal algorithm are written in Java and hence are executed by a "real", i.e. commercially available, JVM. The output of the model checker is whether or not the invariant holds, or the program is deadlock free — if the result is negative, a counter example is produced that can be fed back to the MC-JVM in a simulation mode in order to recreate the error. Next, we briefly highlight some of the novel features of MC-JVM:

**Canonical Heap Representation:** Regardless of which interleaving of statements is executed, dynamic (and static) memory will always be allocated in the same memory locations as they were during any previous interleaving. This feature is related to symmetry reductions and reduces the size of the state-space at the cost of a small run-time overhead.

**Garbage Collection:** The MC-JVM has its own garbage collection scheme, since without it many Java programs would have states that grow indefinitely, which ultimately means the state-space would be infinite.

**Nondeterminism:** Java does not support nondeterminism directly, but in order to do model checking one often wants a nondeterministic *environment* to drive the program under investigation. The MC-JVM traps certain method calls, Verify.Random(i) and Verify.RandomBool(), that respectively returns a nondeterministic integer value between 0 and $i$, or, *true* or $false$ in order to allow nondeterminism.

**Atomicity:** The level of atomicity during execution can be set to be one bytecode instruction, the bytecodes for one Java instruction, the bytecodes for one line of Java code or, lastly, all the bytecodes in a block of Java code that are independent with respect to any code that could be executed concurrently with it.

**Structured State:** Each state of the program under investigation within the MC-JVM is highly structured, since it is in fact consisting of a number of different Java classes. Most explicit state model checkers store states in a flat, state-vector style — we believe the structure in the MC-JVM states can allow more efficient storage.

[1] We were not aware of these systems until late in the development, but as it turned out we made similar design decisions to those made by the JIJ developers.

## 3 Abstraction

We have been developing an automated abstraction tool, which converts a Java program to an abstract program with respect to user-specified abstraction criteria. The user can specify abstractions by removing variables in the concrete program and/or adding new variables (currently the tool supports adding boolean types only) to the abstract program. Specifically, the user selects variables that must be removed and adds abstract variables that represent the predicates in which these variables occurred (typically the predicates are selected from conditions in `if` and `while` statements). Given a Java program and such abstraction criteria, the tool generates an abstract Java program in terms of the new abstract variables and unremoved concrete variables. To compute the conversion automatically, JPF uses a decision procedure, SVC (Stanford Validity Checker), which checks the validity of logical expressions [1].

The abstraction tool is designed for object-oriented programs. The user can specify abstraction criteria for each class by removing field variables in the class and/or adding new abstract variables to the class. Therefore, it can be used for abstracting subcomponents in a program when the whole program is too complicated to apply abstraction globally. In addition, the user can specify new abstract variables which depend on variables from two different classes (inter-class abstraction). There has been similar work by others [5, 14] in different contexts, all of which require use of only global variables to describe a system in simple languages similar to guarded commands. However, our tool targets a real programming language, Java, and is able to handle many problems caused by the object-orientedness [18].

## 4 Static Analysis

Static slicing is a program reduction technique with a wide range of applications in software engineering [17]. JPF uses it to alleviate the state explosion problem during model checking by, first, computing executable program slices that constitute safe abstractions of the original program, and second, identifying which program statements can safely be executed in parallel.

Static analysis has already been applied to model checking. In [13], Millett and Teitelbaum slice PROMELA programs, and in [4], Clarke et al. perform dependence-based slicing on VHDL programs. JPF uses the slicing tool of the BANDERA toolset which implements the work of Hatcliff et al. [7, 6] on static slicing of concurrent Java programs. Their technique consists of computing a set of program dependencies affecting the slicing criteria. These dependencies include the traditional dependencies (data, control and divergence) for sequential programs as well as their counterparts (interference, synchronization and ready dependencies) for concurrent programs. Note that slicing criteria (i.e., program points) are automatically extracted from the properties verified by JPF.

JPF also uses static analysis to compute information to perform partial order reduction during model checking. The static analyzer uses program dependencies to identify safe program blocks. Informally, a safe block is a block of consecutive Java statements that can be executed by the virtual machine without worrying about interleaving, e.g., a sequence of statements using only local variables. Safe blocks are used by MC-JVM to compute "mega" steps in a thread and to perform partial order reduction [12].

## 5 Runtime Analysis

Runtime analysis is conceptually based on the idea of executing the program once, while observing the generated execution trace to extract various kinds of information. This information can then be used to predict whether other different execution traces may violate some properties of interest (in addition of course to demonstrate whether the generated trace violates such properties). Note that the generated execution trace itself does not have to violate these properties in order for their potential violation in other traces to be detected. It's like looking for the footprints of the bug rather than looking for the bug itself. These algorithms typically will not guarantee that errors are found since they work on an arbitrary trace. They also may yield false positives. What is attractive about such algorithms is, however, that they scale very well, and that they often catch the problems they are designed to catch.

An example is the data race detection algorithm Eraser [15], which we have implemented in JPF. A data race occurs when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous. The program is data race free if for every variable there is a nonempty set of locks that all threads own when they access the variable. This is checked by the algorithm. Another example of a runtime analysis algorithm is the locking order analysis, which we have developed and implemented. The locking order algorithm looks for potential deadlocks by detecting differences in the order in which treads take locks. A classical deadlock situation can occur when one thread accesses two Locks in one order, while another thread accesses them in the reverse order. The algorithm searches for the violation of such orderings between locks.

Runtime analysis can be used in two modes within JPF. It can first of all be used stand-alone in simulation mode. Second, runtime analysis can be used to guide the model checker. We have made experiments where the Eraser module in JPF generates a so-called *race window* consisting of the threads involved in a race condition. The model checker is then launched, focusing on the race window by forcing the scheduler always to pick threads in the window before other threads. A dynamic dependency analysis can in addition be used to extend the window with threads that write to objects read by threads in the original window. This resembles dynamic slicing.

## 6 Example : The Remote Agent

The Remote Agent (RA) is an AI-based spacecraft controller that has been developed at NASA Ames Research Center. It consists of three components: a Planner that generates plans from mission goals; an Executive that executes the plans; and finally a Recovery system that monitors the RA's status, and suggests recovery actions in case of failures. The Executive contains features of a multi-threaded operating system, and the Planner and Executive exchange messages in an interactive manner. Hence, this system is highly vulnerable to multi-threading errors. In fact, during real flight in May 1999, the RA deadlocked in space, causing the ground crew to put the spacecraft on standby. The ground crew located the error using data from the spacecraft, but asked as a challenge our group if we could locate the error using model checking. This resulted in an effort described in [8], and which we shall shortly describe in the following. Basically we identified the error using a combination of code review, abstraction, and model checking using JPF1, the first generation of Java PathFinder. During code review we got a suspicion about the error since it resembled one discovered using the SPIN model checker before flight [9]. The modeling therefore focused on the code under suspicion for containing the error. What we will describe in the following is the application of abstraction, runtime analysis and static analysis to reduce the state-space in order to find the above mentioned error in the RA.

The major two components to be modeled were events and tasks, as illustrated in Figure 2. The figure shows a Java class `Event` from which event objects can be instantiated. The class has a local counter variable and two synchronized methods, one for waiting on the event and one for signaling the event, releasing all threads having called `wait_for_event`. In order to catch events that occur while tasks are executing, each event has an associated event counter that is increased whenever the event is signaled. A task then only calls `wait_for_event` in case this counter has not changed, hence, there have been no new events since it was last restarted from a call of `wait_for_event`. The figure shows the definition of one of the tasks. The task's activity is defined in the `run` method of the class `Planner`, which itself extends the `Thread` class, a built-in Java class that supports thread primitives. The body of the `run` method contains an infinite loop, where in each iteration a conditional call of `wait_for_event` is executed. The condition is that no new events have arrived, hence the event counter is unchanged.

```
class Event {
  int count = 0;

  public synchronized void wait_for_event() {
    try{wait();}catch(InterruptedException e){};
  }

  public synchronized void signal_event(){
    count = count + 1;
    notifyAll();
  }
}

class Planner extends Thread{
  Event event1,event2;
  int count = 0;

  public void run(){
    count = event1.count;
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      /* Generate plan */
      event2.signal_event();
    }
  }
}
```

**Fig. 2.** The RAX Error in Java

## 6.1 Applying Abstraction

The shown program has theoretically infinitely many reachable states due to the repeated increment of the count variable in the events. We use abstraction to remove those `count` variables by specifying `Abstract.remove(count)` in the classes of `Event` and `Planner`. In place of these variables, we declare abstraction predicates corresponding to those predicates in the program that involve `count` variables. For instance, we put `Abstract.addBoolean("EQ",count==event1.count)` in the definition of the `Planner` class. After having annotated the program with these abstraction declarations, the abstraction tool is applied and a new abstracted program is generated. JPF thereafter reveals the deadlock in this abstracted program. The error trace shows that the Planner first evaluates the test "`(count == event1.count)`", which evaluates to true; then, before the call of `event1.wait_for_event()` the Executive signals the event, thereby increasing the event counter and notifying all waiting threads, of which there are none. The Planner now unconditionally waits and misses the signal. The solution to this problem is to enclose the conditional wait in a critical section such that no events can occur in between the test and the wait. In fact, the same pattern occurred in several places and in all other places there was such a critical section around. This was simply an omission.

### 6.2 Applying Runtime Analysis

The abstract Java model of what happened on board the spacecraft was created based on a suspicion about the source of the error obtained during code review. This suspicion was created by the fact that this same pattern had been found to cause errors in a different part of the RA during the pre-flight effort using the SPIN model checker two years before [9]. The source of the error, a missing critical section, could, however, have been found automatically using the Eraser data detection algorithm. The variable `count` in class `Event` is accessed unsynchronized by the Planner's run method in the line: "`if (count == event1.count)`", specifically the expression: `event1.count`. Hence even though the `signal_event` called by the Executive will increase the variable synchronized, the above condition in the Planner can be executed even during such a signal. This may cause a data race where the `count` variable is accessed simultaneously by the Planner and the Executive. When running JPF in Eraser mode, it detects this race condition immediately. This could be enough to locate the error, but only if one can see the consequences. The JPF model checker, on the other hand, can be used to analyze the consequences.

To illustrate JPF's integration of runtime analysis and model checking, the example was made slightly more realistic by adding extra threads that made the Java program resemble the real system. The new program had more than $10^{60}$ states. Then we applied JPF in its special runtime analysis/model checking mode. It immediately identified the race condition using the Eraser algorithm, and then launched the model checker on a thread window consisting of those threads involved in the race condition: the Planner and the Executive, locating the deadlock - all within 25 seconds. As an additional experiment in collaboration with the designers of the BANDERA tool [6], we fed part of the result of the race detection, namely the variable that is accessed unprotected, into BANDERA's slicing tool, which in turn created a program slice where all code irrelevant to the value of the counter had been removed. Our model checker then found the deadlock on this sliced program. This illustrates our philosophy of integrating techniques from different disciplines: abstraction was used to turn an infinite program into a finite one, runtime analysis was used to pinpoint problematic code, slicing was used to reduce the program, and finally the model checker was launched to analyze the result.

## 7 Future Work

We will continue to improve the efficiency of the Java Virtual Machine, and the core model checking engine. This involves further development of partial order reduction techniques, and optimized memory management. We intend to further integrate our tools with the BANDERA toolset to achieve an efficient abstraction framework. Runtime analysis is a technique that scales very well, and we intend to study new kinds of errors that can be detected this way. This also involves studying the relationship between static analysis and runtime analysis. As a more ambitious goal, we would like to integrate various kinds of standard program analysis techniques, such as abstract interpretation. We are currently in the final stages of a parallel version of JPF that runs on multiple workstations. In the short-term we will extend JPF to do full LTL model checking, and a long-term project is the development of a more flexible specification language for Java.

## References

1. C. Barrett, D. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, November 1996.
2. D. Basin, S. Friedrich, J. Posegga, and H. Vogt. Java Bytecode Verification by Model Checking. In *CAV '99: 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, 1999.
3. D. L. Bruening. Systematic Testing of Multithreaded Java Programs. Master's thesis, MIT, 1999.
4. E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. Technical Report CMU-CS-99-103, CMU, School of Computer Science, 1999.
5. M. Colón and T. Uribe. Generating Finite-state Abstractions of Reactive Systems using Decision Procedures. In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *LNCS*, July 1998.
6. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering (to appear)*, Limeric, Ireland., June 2000. ACM Press.
7. J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proc. of the 1999 Int. Symposium on Static Analysis*, 1999.
8. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop (to appear)*, June 2000.
9. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998. To appear in IEEE Transactions of Software Engineering.
10. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
11. K. Havelund and J. Skakkebaek. Practical Application of Model Checking in Software Verification. In *Proceedings of the 7th Workshop on the SPIN Verification System*, volume 1680 of *LNCS*, Toulouse, France., September 1999.
12. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.
13. L. I. Millett and T. Teitelbaum. Slicing Promela and its Application to Model Checking, Simulation, and Protocol Understanding. In *Proceedings of the 4th International SPIN Workshop*, 1998.
14. H. Saïdi and N. Shankar. Abstract and Model Check while you Prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454, July 1999.
15. S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
16. A. Taivalsaari. Implementing a Java Virtual Machine in the Java Programming Language. Technical Report SMLI TR-98-64, Sun Microsystems Laboratories, March 1998.
17. F. Tip. A Survey of Program Slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
18. W. Visser, S. Park, and J. Penix. Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice (to appear)*, Auguest 2000.